University of Bucharest
Faculty of Mathematics and Computer Science
Computer Science Specialization

Bachelor's Degree Thesis

3D Real-Time Strategy Game "Guilds"
Made in Unity

Scientific Coordinator

Lect. Dr. Iulia Banu Demergian

Graduate

Simionescu Codruț Andrei

Bucharest, June 2021

Abstract - English

In the present, video games have become one of the most common forms of entertainment, being loved by people of all ages. One of the most popular video game categories is the real-time strategy (RTS) genre. Games belonging to this genre determine the players to make decisions quickly and to think in advance about the tactics they want to use to reach victory.

This paper aims to offer a presentation, as well as documentation for the “Guilds” game demo, developed by myself, which is part of the real-time strategy video game genre. The object of the game is to defeat the enemy units before they defeat the player’s units. In order to succeed in victory, the player must quickly develop a plan, and use the villagers at his disposal at the beginning of the level to gather resources and be able to form his own army which can be led to victory.

It takes good planning and strategic thinking to complete the game, traits that can be developed through exercise, and I believe that strategy games provide such an exercise, thus being a great help for our personal development while also offering us a pleasant way to relax.

I developed the game with the help of the Unity game engine, which offers many tools and integrated systems that help in making a game from scratch, even for a beginner in the field, by having an easy-to-use interface and providing many easy-to-access tutorials on the internet. The scripts used in the project are written in C#, being the programming language integrated with Unity for providing functionality to the objects within the developed game’s scenes.

Due to the fact that developing a complete real-time strategy game requires a lot of time and work, in the current stage, the game “Guilds” is still a demo. offering us a look at the features of this type of game. However, having built a strong foundation with the core mechanics fully set up, I believe that it showcases a huge potential for the video game world of strategy games, and will be proving itself in the upcoming versions.

Abstract - Limba română

În prezent, jocurile video au devenit una din cele mai comune forme de agrement, fiind îndrăgite de către persoane de toate vârstele. Dintr-una din categoriile de joc cele mai populare fac parte jocurile de strategie în timp real (RTS). Acestea determină jucătorii să ia decizii într-un timp scurt și să se gândească din timp la tactica pe care vor sa o abordeze pentru a ajunge la victorie.

Aceasta lucrare are ca scop prezentarea și documentarea demo-ului pentru jocul "Guilds", dezvoltat de către mine, care face parte din categoria jocurilor video de strategie în timp real. Scopul jocului este de a învinge unitățile inamice înainte ca acestea să învingă unitățile jucătorului. Pentru a reuși sa obțină victoria, jucătorul trebuie sa își pună repede la punct planul, și să folosească oamenii de care dispune la începutul nivelului pentru a aduna resurse și a putea să își formeze propria armată pe care să o conducă spre victorie.

Este nevoie de o buna planificare și gândire strategică pentru finalizarea jocului, trăsături ce pot fi dezvoltate prin exercițiu, iar eu consider ca jocurile de strategie oferă un bun astfel de exercițiu, fiind, astfel, de mare ajutor pentru dezvoltarea noastră personală, cât și pentru a ne oferi un mod plăcut de relaxare.

Am realizat dezvoltarea jocului cu ajutorul platformei de dezvoltare a jocurilor Unity, care oferă multe instrumente și sisteme integrate ce ajută enorm la realizarea unui joc de la zero, chiar și pentru un începător în domeniu, având o interfață ușor de utilizat și punând la dispoziție multe tutoriale ușor de accesat legate de utilizarea acesteia pe internet. Script-urile folosite în cadrul proiectului sunt scrise în C#, fiind limbajul de programare integrat in Unity pentru oferirea funcționalității obiectelor din cadrul scenelor jocului dezvoltat.

Deoarece dezvoltarea unui joc de strategie în timp real complet necesită foarte mult timp și muncă, în stadiul actual, jocul "Guilds" este la nivel de demo, oferind o privire la noțiunile de bază ale acestui tip de joc, însă, având fundația și mecanicile principale finalizate, consider că dă dovadă de potențialul pe care îl are pentru a face parte din lumea acestei categorii de jocuri, și pe care îl poate fructifica în cadrul viitoarelor versiuni.

Table of Contents

List of Figures

I. Introduction

The video game industry has grown exponentially in the last decade. Nowadays, people of all ages use video games as a form of entertainment. I've always been passionate about game development and I enjoyed trying to understand the way games work, and strategy games have always been a favorite, determining me to think quickly and have a carefully developed plan in order to achieve success. For this reason, I chose to create the game "Guilds" as my final project.

"Guilds" is a 3D Real-Time Strategy game in which you control several units, which can be described as a bunch of villagers, and you can order them to go to certain places on the map, gather resources, build, or fight with enemy units. The goal is to beat the computer-controlled opponent by improving your initial base, growing a larger unit force, and, by using tactics, be able to defeat all the enemy forces before they overwhelm you.

The Real-Time Strategy (RTS) genre was always a popular one, originating from very early games. Some of the most popular from the past are Age of Empires, StarCraft, Warcraft III, which received many good reviews from critics and still have a relatively big fan base. Because of their success, they were followed by sequels that were also acclaimed by the fans, and to this day, the demand for RTS games is just as big.

Moreover, studies have shown that playing strategy-based video games has a positive influence on our brains. One such research study[1], conducted by researchers from Queen Mary University of London and University College London (UCL) has been published in the journal PLOS ONE, and, as stated by study researcher Dr. Brian Glass, of the School of Biological and Chemical Sciences at Queen Mary University of London: "Our paper shows that cognitive flexibility, a cornerstone of human intelligence, is not a static trait but can be trained and improved using fun learning tools like gaming.", this research offers an in-depth look at the benefits of playing video games from the real-time strategy genre, like StarCraft, which has been used in the study to assess the performance of the volunteer participants on the psychological tests regarding the speed and accuracy in the cognitive flexibility tasks, before and after playing it for a certain period of time. Cognitive flexibility is described as a "person's ability to adapt and switch between tasks, and think about multiple ideas at a given time to solve problems", and I believe that it is an important skill for our brain, and being

able to train it while doing a pleasant activity for us is a wonderful opportunity made possible by the creators of the real-time strategy game titles.

Apart from contributing to the evolution of our fast-thinking skills, there have been people claiming that playing strategy games can also help them relax after a stressful day, concentrating on the tasks presented in the game helping them feel less troubled about the everyday problems in their lives. In order to find out more about this benefit, as well as to confirm the findings of the study presented above, I have gathered data from volunteers willing to provide it through the google forms platform[2]. Here is a summary of what I found by analysing the answers:

- Firstly, I should mention that over half the volunteers who completed the survey have played one or more RTS games.

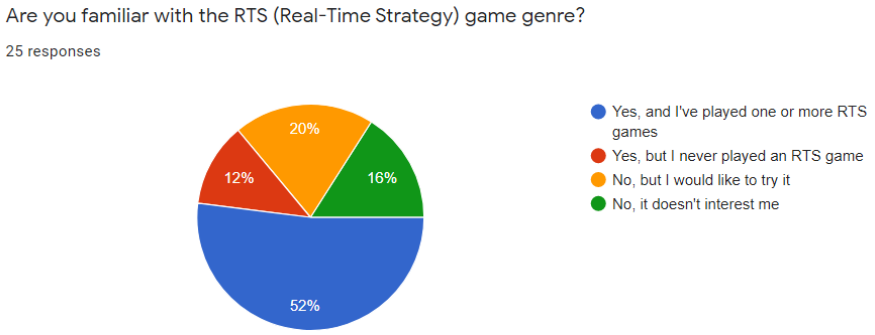

Fig. 1.1. RTS Games Research Q1

- The majority of the volunteers agree that playing RTS games can help the development of fast and critical thinking.

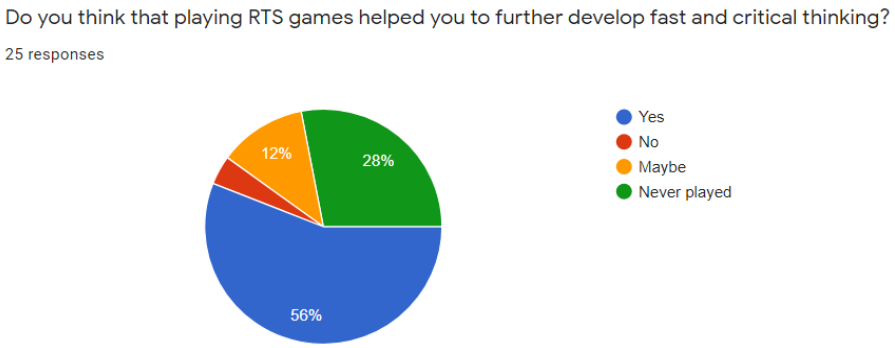

Fig. 1.2. RTS Games Research Q2

- Most of the volunteers who have played RTS games feel relaxed after playing, while some others feel this way only on some occasions.

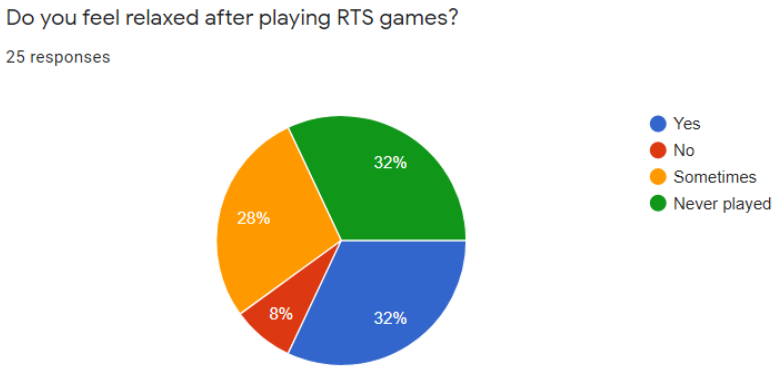

Fig. 1.3. RTS Games Research Q3

- Lastly, half of the volunteers said they would consider playing RTS games in the present, while a third of them said they might do it.

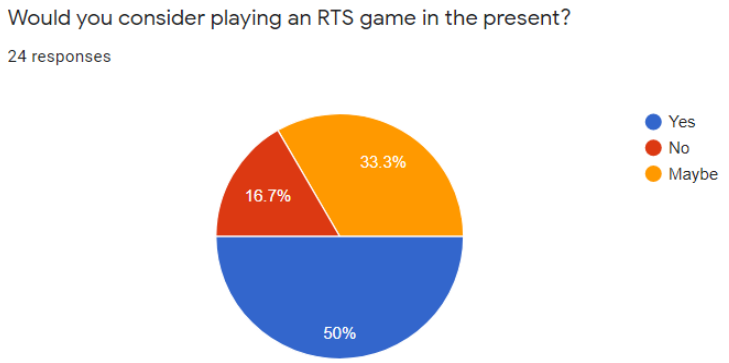

Fig. 1.4. RTS Games Research Q4

There have also been sayings that the real-time strategy genre is dying. However, by taking a look at the responses from the personal research mentioned above, we can see that 50% of the volunteers said they would consider playing an RTS game in the present. Moreover, most recently, Xbox Game Studios announced the release of Age of Empires IV in late 2021, to the delight of its fans, proving the undying desire for the genre, thus making "Guilds" a good fit in today's video game market, having quick thinking and tactical planning, which are acclaimed factors in the genre's success.

The game "Guilds" was developed in the Unity game engine. The most well-known companies that make video games usually have their own game engines which are used for developing their projects. However, lots of games are also being made by teams formed by a small number of developers, even just one person in some cases (like mine). This is why popular game engines like Unity, which can be used by anyone, are very important for the video game industry.

The reason I chose Unity is that it is the most popular game engine that is being used in the present, offering a large variety of tools at the disposal for large teams as well as small ones or individuals wanting to join the industry. Unity works by combining its editor tools with scripts written in C#, a programming language whose object-oriented properties make it very suitable for being used in this environment.

All the assets used for the graphical part of the game, as well as the audio files, were taken from the Unity Asset Store, from multiple creators. The Unity Asset Store offers a large variety of individual assets and asset packs, for free as well as up for purchase, allowing the use of the assets in both personal and commercial ways.

II. Gameplay

II. 1. Main Menu

In its current state, the game is a demo. Developing a complete RTS game requires a lot of time and effort and it is usually made by a team of multiple developers. However, I was determined to take on the journey of developing one from start to finish, and the demo that I will be talking about here, having the core gameplay features finished, is the proof of concept for its promising future.

As soon as the game is built with Unity, it can be started up by opening the executable file. Once opened, the player can see the main menu of the game with the following options, in this order:

- "Continue": When clicked, it loads the last saved game (using a screen fade out loading, followed by the fade in to the loaded level), allowing the player to continue from where he/she left off in the last playthrough. If no save file is available (e.g. if the game was started for the first time), the text is grayed out and the button is unclickable.

- "New Game": When clicked, a new-game sub-menu will pop up, prompting the player to pick the colors for his/her units and for the enemy units from a preset list of colors (the colors have to be different from each other). Once ready, the player can click the "Start game" button, which prompts the screen to fade out to a green color and start loading the first level of the game. Once the scene has finished loading, the screen will fade in to the loaded level.

- "Options": When clicked, the general options sub-menu will appear on-screen. The game options sub-menu will be further detailed below.

- "Quit": When clicked, the application will close, returning to the desktop.

Fig. 2.1. Main Menu

The general options sub-menu appears on top of the main menu and its role is to allow the user to change the following game settings:

- Window Resolution (based on the player's monitor's available display resolutions)
- Full-Screen Mode
- Graphics Quality
- Master Volume

When changed, the new settings take effect immediately. The preferences are saved using the PlayerPrefs system and will remain the way they were set even if you close and reopen the game. The sub-menu can be closed via the "Back" button, located at the bottom, under the settings options.

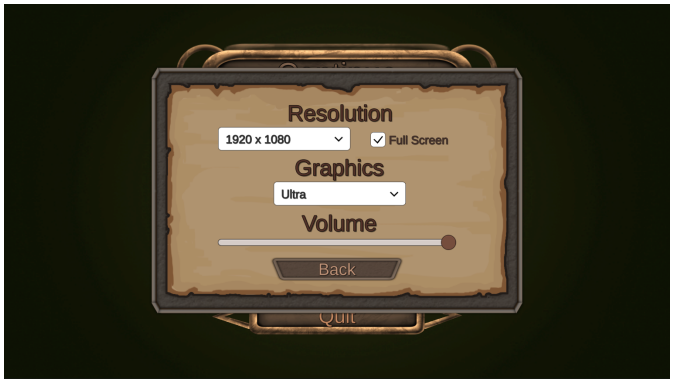

Fig. 2.2. General Options Sub-menu

II. 2. Level Map

The levels of the game take place on a 3D map which consists of:

- Fields, hills, and mountains, with various textures being used for the ground
- Multiple kinds of trees using different models
- Resource fields of different types, further detailed below in section II. 4. Resources
- Units and buildings, belonging to either of the teams in play (player or enemy), further detailed below in sections II. 5. Units and II. 6. Buildings

The map's terrain was made using Unity's terrain tool, which allows for easy building of various landscapes with different altitudes and simple tree placement. Using different textures, I made the map dispose of multiple terrain types e.g. portions of grass, sand, muddy or rocky. The trees are scattered all around the map but are mostly placed in high density along the map's sides, to allow for player activity throughout the map's inner part.

In order to limit the playing field, and set boundaries on the map, there are mountains placed all around the maps' edges. They serve to block the player from viewing anything outside of the map, as well as to prevent the units from moving outside of the level space, no unit being able to climb the mountains used as boundaries.

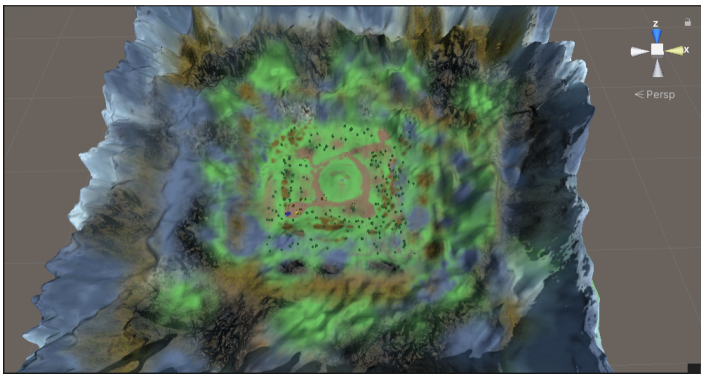

Fig. 2.3. Level Map Boundaries

When starting a new game, the first area the player is able to see is the starting player base, which is located in the lower-left corner of the map. It consists of a few villagers, some resource fields of each type, and a resource camp for each type of resource. Around the map, however, there are groups of enemies scattered, consisting of up to 5 units. More resource fields can also be found scattered around the map.

Fig. 2.4. Starting player base

The whole map can be fully seen at any time on the minimap, which will always be located in the lower right corner of the screen. The minimap's functionality is achieved using an orthographic camera located above the center of the map, looking directly downwards. The edges of the player's vision projected on the map's surface are also displayed on the minimap, being updated constantly as the player is navigating around the level. The minimap rotates along with the main camera.

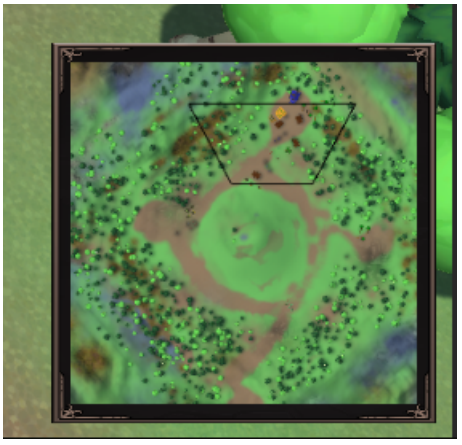

Fig. 2.5. Minimap Close-up

The player can also interact with the minimap by clicking on it. By left-clicking a spot on the minimap, the main camera will move to cover that location on the level's map, and by right-clicking, provided there are units selected, they will be ordered to move in the clicked spot on the level map.

The position of the minimap on the screen has been chosen with the help of a poll, which I had made in order to get opinions from the RTS genre's fans. The results can be seen here[3], as well as in the picture below, showing that a right-sided minimap was the most

preferred option among the possible players. Following this advice, I chose to position the minimap in the lower-right corner.

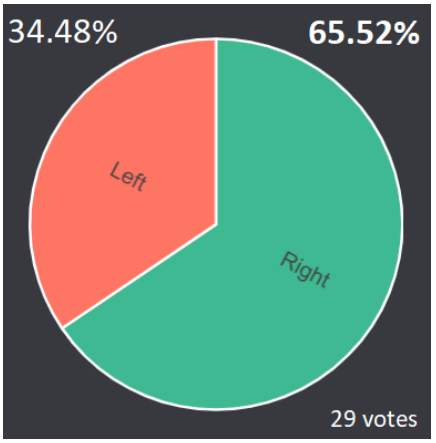

Fig. 2.6. Minimap Position - Poll Results

II. 3. Camera

The way the player can view the level map and effectively play the game is through a floating perspective camera, located above the map's surface, and aimed towards the playing field at an angle between 60 and 90 degrees (60 degrees by default), with a field of view of between 45 and 75 degrees (60 degrees by default).

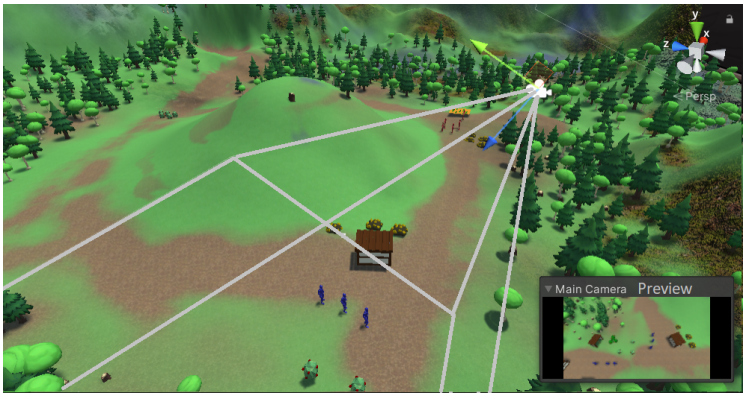

Fig. 2.7. Main Camera Perspective

The camera can be controlled through both keyboard and mouse. It can be moved by either using the WASD keys (W = up, S = down, A = left, D = right) or by moving the mouse cursor near the screen's edges, prompting the camera to move in the corresponding direction. It can move horizontally (by pressing A or D or moving the mouse cursor to the left or right

screen edge), vertically (by pressing W or S or moving the mouse cursor to the top or bottom screen edge), or diagonally (by pressing a combination of the vertical and horizontal keys or moving the mouse cursor to one of the corners of the screen).

The movement of the camera is smoothed, starting at a faster speed and gradually lowering before stopping completely when the movement command is no longer detected. The same mechanic is also valid for camera rotation.

The player can rotate the camera (on the Y-axis), clockwise or counterclockwise, by pressing the Q key (for counterclockwise rotation) or the E key (for clockwise rotation). The rotation can happen in 2 possible ways:

- Progressive rotation: the camera is rotated at slow speed as long as the player is holding down the Q or E keys. This allows the player to set the camera's rotation to any degree value, by pressing either of the hotkeys, and holding it until the desired rotation is achieved.

- Snap rotation: the camera is rotated to snap values, by modifying its rotation with increments or decrements of 90 degrees each time the Q (decrement) or E (increment) keys are pressed, meaning the camera will always be rotated to either 0, 90, 180 or 270 degrees. This allows the player to very quickly rotate the camera to any of the 4 possible perspectives.

The rotation of the camera is performed relative to the center of the screen, keeping it in the same place while rotating, to prevent the player from losing the focus from the currently viewed area.

The minimap camera will always rotate along with the main camera, in order to reflect the same perspective on the level map, keeping the cardinal points oriented in the same way, as to not confuse the player when the camera is rotated.

Both the camera movement and the camera rotation can be sped up by the player by pressing and holding the Shift key while moving or rotating the camera. Keeping the Shift key pressed multiplies the speed at which the camera moves and rotates by 2, allowing for faster camera adjustments.

It is also possible for the camera to be zoomed in or out on the map using the mouse scroll wheel. When zooming in, the camera will advance forwards towards the spot where the

mouse cursor is positioned on the map, at the time of scroll, making it easier for the player to zoom in directly on a specific portion of the map without the need to move the camera first.

However, when zooming out, the camera will move backwards in a straight line, keeping the screen center in the same place, in order to give a wider view of the surrounding landscape and to avoid unwanted situations where the player has the mouse cursor in a corner of the screen when zooming out, making the camera zoom out in a weird, unwanted manner.

Having the zoom-in and the zoom-out work in this way also creates the possibility for the player to navigate the map more easily, by zooming in and out in any direction, allowing for an alternative form of movement for the camera.

II. 4. Resources

There are 3 main types of resources in the game: food, wood, and gold. The player has to collect resources from each type in order to be able to build new units and buildings and be able to finish the game levels.

The amount of resources from each type that the player has accumulated so far in a level can be viewed on the bar at the top of the screen, each resource type having its own infobox.

In order to gather resources, the player has to use the villagers to harvest from the resource fields (which can be found near the initial base, as well as scattered on the map), and bring the harvested materials to a corresponding Resource Camp (further detailed in section II. 6. Buildings). Once they are brought to a resource camp, the info box for that resource type will update the amount of resources of that type that the player has, adding the newly harvested amount.

Even though there are only 3 main types of resources, there are more types of resource fields. This is because there might be multiple sources for acquiring one of the three types. In this demo, this is only valid for food. The resource field types available in the demo are:

- Berry Bushes - Small & Large (food)
- Farms - Small & Large (food)

- Fallen Trees (wood)
- Gold Ores (gold)

Fig. 2.8. Resource Fields

Each resource field can have different properties (for example, the harvesting time might be bigger for farms than for berry bushes). Also, villagers have different carry speeds when carrying different types of resources (for example, villagers carrying gold ore will move slower than villagers carrying berries).

All the resource fields found on the map have a limited number of resources that can be harvested from them. Once that amount is exhausted, the field will disappear, preventing further harvest, determining the player to search for different fields on the rest of the map.

Aside from being stored in a resource camp, carried resources can be dropped on the ground, where they can be left until picked up again by the same or by a different worker.

II. 5. Units

Units are the playing avatars used by the player and the enemy AI to play the game. They can move across the map and perform tasks depending on the type of unit. Each unit may have one or more of the following functions:

➢ *Unit Function*: Every unit has the unit function. It allows the unit to have a current and a maximum health, be controlled by the player or the enemy AI,

be moved around the map, be attacked by fighter units from the opposing team, and die if their life is completely spent.

- ➢ *Worker Function*: A unit with the worker function can harvest resource fields, carry resources around the map, and construct buildings (further detailed in section II. 6. Buildings). When commanded to harvest a resource field, a cycle is started, where the unit keeps going to the resource field, harvesting it, taking the gathered resource to the closest resource camp, and returning to the resource field to repeat the previous steps until ordered to stop or the resource field becomes exhausted, at which point they will search for another resource field of the same type in the former field's surroundings When ordered to collect a different type of resource than the one they are carrying at the moment, workers will drop currently carried resource before starting the new task.

- ➢ *Fighter Function*: Having the fighter function allows a unit to be used in combat against enemy units. It can attack, damage, and kill units belonging to the enemy team. Fighter units will attack nearby enemy units automatically if they are in close proximity. When they do so, they can also alert nearby allied units to attack the hostile unit together. When attacking enemy units on their own (not ordered by the player), they will only chase them for a certain distance (if the enemy unit runs from battle), and then return to their previous position.

The player can select one (by clicking on it) or more (by clicking and dragging the cursor over an area) units belonging to him and order them to move or perform tasks. When moving several units, they will move in an organized manner, arranging themselves in a formation, with the units closest to the front of the new formation moving first, avoiding obstructing themselves, and orienting the formation to face the direction of movement. All units of a certain type currently visible on the screen can also be selected by double-clicking on any of them. The tasks that can be ordered, aside from moving, are gathering resources, storing resources, constructing a building, dropping or picking up a harvested resource, and attacking an enemy unit.

The main units in the game and the only units available in this game demo are villagers. Usually, in these kinds of games, there are different specialized units for harvesting

resources and fighting. However, in “Guilds”, villagers are a multifunctional kind of unit. They can be used for resource gathering as well as fighting, having both the worker and the fighter functions along with the unit one.

Fig. 2.9. Villagers

Killing all enemy villagers will result in a player victory for the current level and a “victory” message written in the player’s color will be displayed on the screen. However, losing all your villagers will result in a player loss for the level, and a victory for the enemy team, triggering the appearance of a “defeat” message with the enemy’s color.

To make the game more challenging, waves of enemy villagers will start appearing on the map two minutes after the level starts and continue spawning at intervals of 60 to 90 seconds until the player manages to beat the game by killing every enemy unit on the map. The first wave of attacking enemies only has 1 enemy villager, while the second wave has 2 enemy villagers, and so on until the fifth wave, which has 5 enemy villagers, and so do all the waves that come after that one. After appearing on the map, the enemy villagers will proceed to chase and attack the closest unit (belonging to the player team) to them. If their target is killed and there are no other player-owned units nearby, they will remain in that same spot, guarding the area by attacking any player-owned units getting too close. In this scenario, the remaining group also has to be killed for the player to achieve victory.

Being a demo, there are no additional unit types present in the current game version, but the functions used by the villagers can very easily be employed by any additional unit type to be implemented in the future versions.

II. 6. Buildings

Buildings are types of structures found in the game serving different purposes. The first buildings the player can notice in the game are the initial resource camps which are found in the starting player base. New buildings can be constructed by villagers if the player has the required amount of resources needed.

Resource camps are one of the two buildings found in this demo. Their purpose is to provide a way for the player to store gathered resources. When starting a new game, 3 resource camps are already built in the initial base (one for each resource type). Additional resource camps can be built by villagers at the cost of 50 wood, allowing the player to harvest distant resource fields more easily. When newly built, the camps will have no resource type assigned, making it possible for any type of resource to be stored within. However, once a resource type is assigned to it (by storing a number of resources of that type in the camp), it cannot be changed to a different type later. Resources can only be stored within their corresponding type of resource camp or in an empty one.

Besides resource camps, the player starts with a *Villager Inn*. It is the most important building of the player because it can produce more villager units (through the recruit button on the interaction panel in the bottom-left corner), at the cost of 45 food and 35 gold per Villager. Constructing additional Villager Inns is more expensive resource-wise, requiring 150 food, 150 wood, and 225 gold for its construction. Villager Inns will always have the color of their team displayed on the roof.

Fig. 2.10. Buildings

In order to construct additional buildings (by using villagers), the building has to be selected from the villager's build panel (located in the interaction panel in the bottom-left

corner). Once selected, the player has to choose a place where the building construction will begin, by placing a preview of that building on the map. Previews can also be rotated, altering the rotation of the future building, by using the Q and E keys (Q - counterclockwise rotation, E - clockwise rotation; the camera cannot be rotated while previewing a building construction).

Constructions can only be started in unobstructed places (meaning they cannot be placed over units, resource fields, trees, or other buildings), where the ground elevation does not have big altitude differences over the base of the constructed building. If the aforementioned conditions are met, the color of the preview building turns green, otherwise (when it's incorrectly placed) it turns red.

As long as the color of the preview is red, the player will not be able to start the construction in that location. If the player clicks on a spot when the color is green, the building construction will start, and the necessary resources for the construction will be deducted from the total amount available to the player. Multiple villagers can construct the same building at the same time, speeding up the time required to finish the construction.

II. 7. Pause Menu

When playing a level, the player can use the "Menu" button, located in the top-right corner of the screen, to pause the game and bring up the pause menu. The same result can be achieved by pressing the ESC key while the game is unpaused.

When the pause menu is brought up, the time in the game becomes completely frozen, with all the units remaining in the same position and state. The menu consists of the following option buttons, in this order:

- "*QuickSave*": The state of the level can be saved using this button. Trying to save multiple times will delete the previously saved file. Every part of the current level state (units, tasks, buildings, resource fields, and drops, along with their specific details) will be saved locally, offering the player the possibility of quitting the game without losing all the progress made in that session.

➢ "*QuickLoad*": This button allows the player to load the previously saved file (created by the QuickSave option), restoring every former unit, building, resource field, and resource drop to the state in which they were in when the level was saved. If no save file exists at the time the button is clicked, the pause menu will close, resuming the game, and a message will appear on the screen announcing to the player that no save file was found.

➢ "*Options*": When clicked, the game options sub-menu will appear on-screen. The game options sub-menu will be further detailed below.

➢ "*Main Menu*": By using this button, the player can return to the game's main menu. Unsaved progress will be lost.

➢ "*Resume*": Pressing it closes the pause menu and unfreezes the game time, each unit resuming what it was doing when the game was paused. This can also be done by pressing the ESC key while the game is paused.

Fig. 2.11. Pause Menu

The game options sub-menu appears on top of the pause menu and its role is to allow the user to change the following level settings:

- Camera Rotation (between 45 and 90 degrees Slider) - Changing this will adjust the camera inclination towards the map, offering a completely top-down perspective when set to 90 degrees.
- Camera Field Of View (between 45 and 75 degrees Slider) - Adjusts the camera's field of view.
- Snap Rotation (Toggle) - Toggles between Progressive Rotation and Snap Rotation camera modes, described in section II. 3. Camera.
- Camera Mouse Movement (Toggle) - Enables or disables the ability to move the camera by positioning the mouse cursor near a screen's edge.

Just like the settings from the general options sub-menu in the Main Menu screen, changing the values will cause the new settings to take effect immediately, the preferences being saved using the PlayerPrefs system and having them remain in the way they were set after reopening the game. The sub-menu can be closed via the “Return” button, located at the bottom, under the settings options.

Fig. 2.12. Game Options Sub-Menu

III. Development

III. 1. Unity Engine

The Unity game engine, which I have chosen to use for the development of the game "Guilds", has always been a popular choice for game development. Its popularity comes from its easy-to-use graphical interface, which is well suited for beginners as well as experienced developers, and from its support for multiple platforms: Windows, Linux, Android, iOS, and also game consoles like PlayStation, Xbox, Nintendo Switch, along with many others.

In order to understand how Unity works, we have to take a look at some of Unity's main component features, used by developers in the development of their games (more information for each of Unity's features can also be found in the Unity User Manual[4]):

- ➢ *GameObjects*: Based on Object-Oriented Programming (OOP), in Unity, every object that is part of a game scene is derived from the GameObject class. The GameObjects can be created or modified within the editor and can also have different components attached to them. One of the most important components is the *Transform* component. Its role is to store the position, rotation, and scale for the respective GameObject. It is possible to create a *Prefab* of a GameObject which is used to create several instances of the same GameObject in a scene and provides the possibility to modify all of them at once.

- ➢ *Scenes*: Scenes are what every game is composed of. Each level or menu of a game is actually a scene in Unity. A scene is composed of a multitude of GameObjects, set up and planned by the developer, which, put together, make up the content of the game. A game can be made with any number of scenes, even just one for a simple game.

- ➢ *Camera*: Each scene has one or more cameras present in it. Cameras are also GameObjects and their role is to allow the player to see into the game. Scenes in Unity are in three-dimensional space, so the cameras are used in order to "flatten" the GameObjects present in the scene, which are displayed in three-dimensional space, and turn them into a two-dimensional render for the player to see.

➢ *MonoBehaviour*: MonoBehaviour is the link between the editor and the C# scripts written by the developer. It is the base class from which all the Unity scripts derive. It also contains the Unity-specific functions, which are essential for obtaining the behaviour you want for the game features. These functions will be further detailed in section III. 2. MonoBehaviour.

The Unity editor is composed of several window tabs, each with its own purpose. Some of the most used window tabs are:

- *Scene View*: This is where you can edit the scene you are working on, being able to navigate through it, select the GameObjects within, and change the perspective from 2D to 3D and vice versa, depending on your kind of project

- *Game View*: Using this view allows you to see what the player will be seeing when the game will be running, outputting the renders of the cameras in the scene.

- *Hierarchy Window*: It represents the text representation of all the GameObjects present in the scene in a hierarchical way, while also displaying their structure and the way they are attached to each other.

- *Inspector Window*: Using the Inspector window allows you to modify all the properties of the selected GameObject and its components, as well as add any other components or remove some of the existing ones. Its layout changes depending on the selected GameObject's characteristics.

- *Project Window*: Here will be displayed all the assets present in the current project. Every imported asset can be found here. It also helps in finding, modifying. or deleting certain assets and in maintaining an organized project assets hierarchy.

- *Animator Window*: This is where you can control how the animations work for any GameObject in the project. It functions as a state machine where each animation is a state, and by changing the transitions and conditions between the states, you can achieve the desired animations for your game.

III. 2. MonoBehaviour

As stated before, in order to create a script to make something happen the way we want it to in our game, we have to derive that script from the MonoBehaviour class. Aside, from this, all scripts have to be written in the C# programming language.

Deriving from MonoBehaviour allows us to use the Unity-specific functions, required to have our game be played as intended. Some of the ones I used the most in this project are the following:

- *Awake()*: This is called right when the instance of the script is loaded. It is called before Start() and Update() functions, and can be used to initialise the script instance with certain values or states. It is only called once in its lifetime.

- *Start()*: Right after a script is enabled, this is the first function called. It is called before the Update() functions. Just like the Awake() function, it is only called once in its lifetime.

- *Update()*: The Update() function is called on every frame of the game. It can be used in many ways, like checking whether an event has occurred and proceeding accordingly.

- *FixedUpdate()*: This function is similar to the Update() function, but it is only called on each physics frame, having a fixed frame rate (50 times per second by default, managed by the Physics system), which makes it frame-independent. It is usually used for operations that involve Physics calculations.

- *OnDestroy()*: This function is called for each GameObject upon its destruction. It can be called when the scene ends, destroying every GameObject in it, or when the object is specifically destroyed with the Destroy() function.

- *CompareTag(string tag)*: Every GameObject can be assigned a tag, by using this function we can check whether a certain GameObject has the same tag as the one passed to the function.

- *Instantiate(GameObject obj)*: This is used to clone the object obj, returning the clone. It is mostly used with Prefabs, in order to insert objects of the prefab's type into the current scene on runtime.

- *Destroy(GameObject obj)*: You can remove a GameObject or one of its components from the scene by using this function. Removing a GameObject will eliminate it from the current scene, and trigger the OnDestroy() function.

- *OnTriggerEnter(Collider other)*: Unity's Physics System uses *Colliders* to check for collisions between GameObjects having a collider component attached. A collider can be set to *Trigger* to avoid stopping other objects from passing through it. This function is called whenever an object with a collider attached enters the area of the trigger collider attached to the GameObject that has a script using this function.

- *StartCoroutine(IEnumerator routine)*: Normally, functions begin and finish their execution in the same frame. However, there are cases where you want to prolong the execution's duration, in order to achieve a result over the course of more than one frame. For this reason, *Coroutines* can be used. A coroutine is a function that can stop its execution for a certain time, using the *yield* statement, be it just for the remainder of the current frame, or for a specified number of seconds, allowing us to achieve the desired result over several frames. Coroutine functions, which are declared as IEnumerator, can be started using the StartCoroutine() function.

Having clarified some of the advantages offered by deriving our classes from MonoBehaviour, we can now take a look at how the "Guilds" game demo was created, in the following sections.

III. 3. User Interface

Working with UI in Unity requires a *Canvas* to be present in the scene. A Canvas is a GameObject that covers up the whole screen at all times. It can scale in 3 different modes: Constant Pixel Size, Scale with Screen Size, or Constant Physical Size. For most of the canvases in my game, I used the Scale with Screen Size mode, in order to avoid any scaling problems when resizing the game window.

The Main Menu of the game is realised using UI *Buttons*. Buttons have an onClick() function which is managed by the Unity EventSystem, which registers all the input from the user to the UI. The onClick() function can be assigned multiple actions that will be performed when the button is clicked. The actions can be simple, like enabling or disabling a

GameObject, or more complex, requiring a script function to be called when the button is pressed. All such functions used for the Main Menu scene can be found in the MainMenu.cs script.

When the player presses the New Game button. The team color picker sub-menu is enabled by the onClick() function. After the color is chosen from the list of predefined colors, the NewGame() function is called.

```
public void NewGame()
    {
        saveFileExists.saveToBeLoaded = false;
        AssignColors();
        StartGame();
    }
```

The first line tells the SaveLoadManager.cs script (through the saveFileExists variable that is actually a boolean) to prevent the player from quick-loading any save file (any previous save file cannot be loaded when starting a new game). The function on the second line assigns the chosen colors to the materials used by the units from each team. Materials are used by the renderer to display the graphics of the game. The function StartGame() on the third line looks like this:

```
    private void StartGame()
    {
        Instantiate(fadeCoverPanel);
        StartCoroutine(LoadGame());
    }

    private IEnumerator LoadGame()
    {
        yield return new WaitForSeconds(1f); // wait for fade to finish fading

        AsyncOperation asyncOperation = SceneManager.LoadSceneAsync("Main");

        while (!asyncOperation.isDone)
            yield return null;
    }
```

The fadeCoverPanel is a UI Panel whose role is to progressively cover the whole screen (through an animation made using the Unity Animation System that raises the transparency value -alpha- of the panel over the course of 1 second), and keep the screen covered while the scene of the level is loaded. This is done through an asynchronous operation. The coroutine LoadGame() pauses its operation until the fadePanel finishes its animation and then waits for the scene to finish loading. This way, the player will only see the color of the panel while the level is loading. When the level is loaded, there is another fadePanel in the scene of the level, which has the fade animation in reverse, fading from the color of the panel to the map of the level, that is instantiated right at the start, resulting in a

smooth transition between the scenes.

When pressing the Continue button, the ContinueGame() function is called instead of the StartGame(), which calls the StartGame() function too, but only after checking if there is a save file to be loaded and tells the SaveLoadManager through another bool variable to load the file right after the scene of the level is loaded.

```
public void ContinueGame()
    {
        if (saveFileExists.saveToBeLoaded)
        {
            continueSavedGame.saveToBeLoaded = true;
            StartGame();
        }
    }
```

The Options button enables the general options sub-menu GameObject, which is set up in a similar way to the main buttons, by using the EventSystem and some function calls from the MainMenu.cs script. And there's the same case when pressing the Quit button, having the QuitGame() function called by the onClick() function.

```
    public void QuitGame()
    {
        Application.Quit();
    }
```

All the game settings set in the game options sub-menu are saved between sessions using the PlayerPrefs System.

There are also UI elements in the scene of the main level, but they will be acknowledged in the next section, which tackles the implementations of the game's core mechanics.

III. 4. Behind Game Mechanics - Scripts

In this section, I shall explain how the most important aspects of the game work, the thought process behind their implementation, and have a look at the scripts required to make it all possible.

The game has many different mechanics. To program each of the game's features to work the way I wanted, I decided to use several *manager classes*. These classes are made using the *Singleton* type (to make sure there is only one of each manager active at once in a scene) and are attached to the GameManager GameObject in the scene.

For developing this demo, I used the following manager classes:

➢ *III. 4. 1. GameManager.cs*

The GameManager class holds the lists with references to every object in the level, along with the stats for the units and buildings and the reference to the main terrain used by the map. Apart from keeping count of all the objects, it also has the functions required to pause and unpause the game, by freezing the time scale, checks whether the level should end (in a victory for the player or the enemy team), can clear the scene of every entity, and can provide the GameObject located at certain positions to other scripts, like the SaveLoadSystem, which needs to clear the scene, and, after re-instantiating the entities, it resets the targets for each of the units based on the targets' positions. Here are some of its functions:

```
    public void PauseGameState()
    {
        isPaused = true;
        Time.timeScale = 0f;
    }

    private int GetNumberOfUnitsOfTeam(Team team)
    {
        int units = 0;
        foreach (Unit unit in activeUnits)
            if (unit.unitStats.unitTeam == team)
                units++;
        return units;
    }

    private IEnumerator CheckGameEndCo()
    {
        yield return new WaitForSeconds(1f); // the end condition is checked every 1 sec

        int playerUnits = GetNumberOfUnitsOfTeam(Team.PLAYER);
        int enemyUnits = GetNumberOfUnitsOfTeam(Team.ENEMY1) +
GetNumberOfUnitsOfTeam(Team.ENEMY2) + GetNumberOfUnitsOfTeam(Team.ENEMY3);

        if (playerUnits == 0)
            StartCoroutine(GameOver(false)); // level finished - enemy win
        else if (enemyUnits == 0)
            StartCoroutine(GameOver(true)); // level finished - player win
        else
```

```
            StartCoroutine(CheckGameEndCo()); // level not finished - keep checking
    }
```

➢ *III. 4. 2. CameraController.cs*

The CameraController class manages every aspect of the main camera. It checks for input from the player, and, if the camera has to be moved or rotated based on the received input, it performs the movement or rotation in a smooth manner, keeping count of the bounds imposed for the camera's height and position.

On every frame, the check for input from the player is performed:

```
    private void Update()
    {
        CheckMovement();
        CheckZoom();
        CheckRotation();
    }
```

At the end of the frame (with the help of the Late Update() function), the camera is moved and rotated as needed.

```
    private void LateUpdate()
    {
        MoveCamera();
        CheckBounds();
    }
```

The movement and rotation of the camera are smoothed through the use of the Lerp() function, which linearly interpolates between two points (for movement) or two quaternions (for rotation). The values for desiredCameraPosition and desiredCameraRotation are decided by the functions checking for player input.

```
    private void MoveCamera()
    {
        if (transform.position != desiredCameraPosition)
            transform.position = Vector3.Lerp(transform.position, desiredCameraPosition,
moveSmoothing);

        if (transform.rotation != desiredCameraRotation)
            transform.rotation = Quaternion.Lerp(transform.rotation,
desiredCameraRotation, rotationSmoothing);
    }
```

The bounds of the camera are based on the limits of the main terrain used for the level of the map and on the current height of the terrain at the camera's position. The

Mathf.Clamp() function clamps the value given as the first argument between the two values given as second (minimum value) and third (maximum value) arguments, preventing it from going below or above those values.

```
    private void CheckBounds()
    {
        float currentTerrainHeight =
terrainToHoverOver.SampleHeight(transform.position);

        minHeight = currentTerrainHeight + minHeightFromGround;
        maxHeight = currentTerrainHeight + maxHeightFromGround;

        desiredCameraPosition.y = Mathf.Clamp(desiredCameraPosition.y, minHeight,
maxHeight);

        desiredCameraPosition.x = Mathf.Clamp(desiredCameraPosition.x,
terrainPosition.x, terrainPosition.x + terrainSize.x);
        desiredCameraPosition.z = Mathf.Clamp(desiredCameraPosition.z,
terrainPosition.z, terrainPosition.z + terrainSize.z);
    }
```

➢ *III. 4. 3. SelectionManager.cs*

The SelectionManager class handles the player's ability to select units or buildings. The way this works for single selection (clicking on a unit or building) is by using raycasting. By casting a physics ray from the mouse cursor's position on the screen by using the camera.ScreenPointToRay(Input.mousePosition) function when the player left-clicks over something, the ray will hit the object on the map (if it's on the "Interactable" layer), and return information about it. Then we check if that object is a unit or building from the player team, and select it if the answer is positive, by keeping it in a list, and enabling the "Selected" area around it, so the player will know it has been selected.

For multiple selection (click-and-drag selection), however, things get a bit more complicated. Because we cannot cast rays on every bit of space in our selection, as it would be too inefficient, we have to find another way of gathering information about the objects in our selection area. First things first, we have to determine what our selection area is, more exactly. We need to select every unit in the space between the camera and the four corners of our selection, which can be found by casting rays in the positions of the selection's corners. Although the selection on the screen is a square, the selection space is not going to be a box or pyramid, because the game uses a perspective camera, and the map's terrain is not flat, resulting in an irregular 3D shape. We can set up a flat surface below the map and cast the

rays on a specific layer in order to only hit that surface and solve the height difference problem, but, having the perspective camera, we still can't use the primitive colliders, like the Box or Sphere colliders for detecting the units inside the shape of the selection space. Fortunately, we can use a Mesh Collider. A Mesh in Unity is a shape consisting of triangles arranged in a specific way in 3D space, reassembling a solid object, which means that by creating a Mesh with the same shape as our selection space, we could use the Mesh Collider component to register all the colliders of the units inside of it. We can, then, remove the Mesh on the next frame so that it will only select units on the frame when the player makes the selection. This way, the SelectionManager gets the information about all the units and buildings (even though only one building can be selected at a time and only if no units are selected) found inside the selection area of the multiple selection.

The SelectionManager can also check if the player double clicks on a unit when single selecting, which triggers a multi-selection that selects all the units visible on the screen by simulating a drag selection on the whole screen.

The functions used for performing the multiple selection (except for the player input check and the transparent selection area initialisation, which are performed in the Update() function) can be found in the Annexes section, labeled as **Appendix 1.** *Functions used for implementing drag selection.*

➢ *III. 4. 4. InteractionManager.cs*

The InteractionManager class manages the interactions of the selected units with the target of the order given by the player through right-clicking. It works by checking for player input, and, provided there are one or more units selected by the SelectonManager class, casting a ray from the cursor's position on the screen. The ray is cast on the layers "Interactable" and "Terrain", so the returned hit is either an interactable object or a spot on the map's terrain. Based on the hit object's tag, the corresponding function is then called for each of the selected units, given the hit object as the target.

The possible interaction actions are:

- Harvesting a resource field
- Storing resources in a camp
- Picking up a dropped resource

- Constructing an under-construction building
- Attacking a unit from the enemy team
- Moving to the target position (if no interactable object was hit)

```
    private void Update()
    {
        if (GameManager.instance.IsPaused() || UIManager.instance.IsMouseOverUI() ||
ConstructionManager.instance.IsPreviewingBuilding())
            return;

        if (Input.GetMouseButtonDown(1))
        {
            CheckInteraction();
        }
    }

    private void CheckInteraction()
    {
        if (selectionManager.selectedUnits.Count > 0)
        {
            Ray rayLocation = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(rayLocation, out RaycastHit hitLocation, 1000f,
LayerMask.GetMask("Interactable", "Terrain")))
            {
                PerformInteraction(hitLocation);
            }
        }
    }
```

➢ *III. 4. 5. MovementManager.cs*

The MovementManager class takes care of how multiple units move. Its role is to make the selected units move in an ordered manner, without obstructing each other, and arrange themselves in a tight square formation. On top of this, the formation has to be aligned relative to the center of the previous positions of the units, based on the place where the player right-clicked with multiple units selected.

To achieve this, it checks for the number of rows given the formation is arranged in a perfect square, then it adds part of the remainder to the square sides as much as they fit, and lastly, it puts the ones left at the back of the lines. After this is done, all the positions in the formation are rotated in order to have the formation face the right way.

```
    public void MoveInFormation(Vector3 moveToSpot)
    {
        AssignPositions(GenerateFormation(moveToSpot));
    }
```

The functions AssignPositions() and GenerateFormation() can be found in the Annexes section, labeled as **Appendix 2.** *Functions used for implementing formation movement.*

➢ *III. 4. 6. ResourceManager.cs*

The ResourceManager class handles the resource management in the level. It keeps track of the current amount for each type of resource by using a function that adds a number of resources of a certain type, which is called when resources are stored, and one that subtracts a number of resources of each type or checks whether the player has the required amount of resources, which is called when a unit is recruited or a building construction is started.

The functions AddResources() and UseResources(), used by the manager to control the number of available player-owned resources, can be found in the Annexes section, labeled as **Appendix 3.** *Functions used for managing player-owned resources*.

➢ *III. 4. 7. PrefabManager.cs*

The PrefabManager class has the role of providing references for each prefab used in the level to other classes. It doesn't implement any functions but has references to the prefabs of units, buildings, constructions, resource fields, and dropped resources, which are set to public so they can be accessed by any other classes that can access the PrefabManager class through its static instance.

```
public static PrefabManager instance;

[Header("Units Prefabs")]
public GameObject villagerPlayerPrefab;
public GameObject villagerEnemyPrefab;
public string weaponInHandName;

[Header("Buildings Prefabs")]
public GameObject resourceCampPlayerPrefab;
public GameObject resourceCampConstructionPlayerPrefab;
public GameObject villagerInnPlayerPrefab;
public GameObject villagerInnConstructionPlayerPrefab;

[Header("ResourceFields Prefabs")]
public GameObject berryBushSmallPrefab;
public GameObject berryBushLargePrefab;
public GameObject lumberTreePrefab;
```

```
public GameObject goldOreMinePrefab;
public GameObject farmFieldSmallPrefab;
public GameObject farmFieldLargePrefab;

[Header("Resource Drops Prefabs")]
public GameObject berriesDropPrefab;
public GameObject logPileDropPrefab;
public GameObject goldOreDropPrefab;
public GameObject farmDropPrefab;

[Header("Resource Info")]
public ResourceInfo berriesInfo;
public ResourceInfo woodInfo;
public ResourceInfo goldInfo;
public ResourceInfo farmInfo;
```

➢ *III. 4. 8. ConstructionManager.cs*

The ConstructionManager class handles the construction of new buildings. It implements functions that allow the previewing (and its canceling), checking the placement validity of the construction, rotation of the previewed building, and starting the construction. Every preview building has the PlacementValidity class which checks for collisions from other GameObjects or for the differences in terrain altitude and determines if the construction can begin in the current position.

The building preview's initial rotation is set to be facing the camera (at one of the allowed angles of placement).

The functions used to preview and place building constructions can be found in the Annexes section, labeled as **Appendix 4.** *Functions used for implementing building construction.*

➢ *III. 4. 9. MinimapCameraController.cs*

The MinimapCameraController class rotates the minimap to match the main camera's rotation, in order to prevent confusion for the player when the main camera is rotated. It also draws the outlines of the main camera's visibility of the level's terrain (main camera's frustum) on the minimap, so the player knows what part of the map he is viewing at any time.

```
private void Update()
{
    //We update the main camera frustum on the minimap
    ShowMainCameraFrustum();
```

```
    }

    private void LateUpdate()
    {
        //We rotate the minimap so that it will have the same rotation on the y axis as the main cam
        RotateMinimap();
    }
```

The functions ShowMainCameraFrustum() and RotateMinimap() can be found in the Annexes section, labeled as **Appendix 5.** *Functions used for controlling the minimap*.

➢ *III. 4. 10. MinimapInteractionController.cs*

The MinimapInteractionController class controls the player's input to the minimap's area. When the player left-clicks on the minimap, this class will move the main camera to that spot on the level map, and when the player right-clicks on the minimap with units selected, the class will order the units to perform the right interaction on that indicated spot from the level map. This is done by casting rays through the minimap camera, hitting the spots on the actual level map, similar to the way it happens in the InteractionManager class.

```
    public void OnPointerClick(PointerEventData eventData)
    {
        if
(RectTransformUtility.ScreenPointToLocalPointInRectangle(GetComponent<RawImage>().rectTransform,
eventData.pressPosition, eventData.pressEventCamera, out Vector2 localCursorPoint))
        {
            Rect imageRectSize = GetComponent<RawImage>().rectTransform.rect;

            /* localCursorPoint is the distance on x and y axis from the rect center point off we
add the imageRectSize (by subtracting because it's negative) which is the half size the rectangle so
we can get the local coordinates x and y inside the rectangle then we divide them by the rectSize so
we can get their ratios (between 0.0 - 1.0) */
            localCursorPoint.x = (localCursorPoint.x - imageRectSize.x) / imageRectSize.width;
            localCursorPoint.y = (localCursorPoint.y - imageRectSize.y) / imageRectSize.height;

            CastMinimapRayToWorld(localCursorPoint, eventData.button);
        }
    }

    private void CastMinimapRayToWorld(Vector2 localCursor, InputButton mouseButton)
    {
        //we multiply the local ratios inside the minimap image rect with the minimap camera's
pixelWidth so we can get the right pixel coordinates for the ray
        Ray miniMapRay = minimapCam.ScreenPointToRay(new Vector2(localCursor.x *
minimapCam.pixelWidth, localCursor.y * minimapCam.pixelHeight));

        //we cast the ray through the minimap camera, which will hit the world point that it pointed
towards
        if (Physics.Raycast(miniMapRay, out RaycastHit minimapHit, 1000f,
LayerMask.GetMask("Interactable", "Terrain")))
        {
```

```
            if (mouseButton == InputButton.Left)
            {
                CameraController.instance.MoveCameraToPosition(minimapHit.point.x,
minimapHit.point.z, true);
            }
            else if (mouseButton == InputButton.Right)
            {
                if(SelectionManager.instance.selectedUnits.Count > 0)
                {
                    InteractionManager.instance.PerformInteraction(minimapHit);
                }
            }
        }
    }
```

➢ *III. 4. 11. UIManager.cs*

The UIManager class manages all the user interface elements in the level. The UI elements in the level are:

- The transition from and to the main menu using the fade panel
- The resources infoboxes and the "Menu button at the top of the screen
- The interaction panel used to construct buildings and recruit villagers
- The screen text alert when not enough resources are available or no save file was found
- The Pause Menu and its elements detailed in section II. 7. Pause Menu + animations
- The level-end "victory" or "defeat" displayed text

```
    private void Update()
    {
        UpdateResourceText();

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (GameManager.instance.isPaused)
                UnPauseGameTab();
            else
                PauseGameTab();
        }
    }

    private void UpdateResourceText()
    {
        foodText.text = ResourceManager.instance.currentFoodAmount.ToString();
        woodText.text = ResourceManager.instance.currentWoodAmount.ToString();
        goldText.text = ResourceManager.instance.currentGoldAmount.ToString();
    }

    public void PauseGameTab()
    {
        GameManager.instance.PauseGameState();
```

```
        pauseMenu.SetActive(true);
    }
```

➢ *III. 4. 12. GamePrefsManager.cs*

The GamePrefsManager class makes sure the player preferences for the level settings are saved between sessions. The saved settings are:

- Camera's X-axis rotation
- Camera's field of view (FOV)
- Camera's snap rotation preference
- Camera's movement with the mouse cursor

```
    private void Start()
    {
        LoadPreferences();
    }

    private void LoadPreferences()
    {
        LoadCameraXRotationPref();
        LoadCameraFOVPref();
        LoadCameraSnapRotationPref();
        LoadCameraMouseMovementPref();
    }
```

➢ *III. 4. 13. SaveLoadSystem.cs*

The SaveLoadSystem class provides the player with a way to save their progress in the game, in order to resume the level in the same state it was left when taking a break from the game. It uses a binary formatter to serialize the game data, resulting in a binary save file saved locally, that can be deserialized in order to obtain the previously saved data.

For the game data to be able to be serialized by the binary formatter, it must only contain primitive file types, meaning GameObjects and other Unity-specific classes cannot be used to save the data. For this reason, I created the GameSaveData class which is only composed of primitive types, using float-type arrays to save the positions of the objects in the game, as well as other details specific for each of them. Marking the GameSaveData class as

System.Serializable allows the formatter to serialize the file after all the game data was saved in it (done by the constructor of the data class).

For a save file to be loaded, the process is reversed, at first, the save file is deserialized by using the binary formatter, and, then, all the GameObjects are re-instantiated using the data from the saved file.

All errors that can be thrown by the serialization and deserialization of the save file are caught by try-catch blocks.

The functions SaveGame() and LoadGame() can be found in the Annexes section, labeled as **Appendix 6.** *Functions implementing the save & load game-state features*.

➢ *III. 4. 14. AIManager.cs*

The AIManager class is in charge of spawning the waves of enemy villagers who are attacking the closest player-owned unit to them. It works by starting a coroutine at the beginning of the level, which is pausing its execution until the initial safe time (two minutes) is finished. It, then, starts another coroutine which starts spawning the enemy waves, beginning with the first wave of one enemy villager and pausing its execution until a cooldown, which is a random value between 60 and 90 seconds, passes, followed by the restarting of the coroutine, but with the wave number increased (which also increases the number of enemies spawned, up until 5). The location at which the enemy villagers are spawned is set up as a GameObject in the scene and is manually chosen based on the level's design.

```
private void Start()
{
    StartCoroutine(StartWaves());
}

private IEnumerator StartWaves()
{
    waveNumber = 0;
    yield return new WaitForSeconds(firstWaveStartAfter);
    StartCoroutine(SpawnEnemyWaves());
}

private IEnumerator SpawnEnemyWaves()
{
    waveNumber++;

    for (int i = 0; i < Mathf.Min(waveNumber, maxEnemyUnitsSpawnedPerWave); i++)
```

```
        {
            GameObject enemyUnit = Instantiate(PrefabManager.instance.villagerEnemyPrefab,
enemyWavesSpawnPoint.position, Quaternion.identity,
PrefabManager.instance.unitsTransformParentGO.transform);

            Unit closestPlayerUnit = GameManager.instance.GetClosestUnitOfTeamFrom(Team.PLAYER,
enemyUnit.transform.position);
            enemyUnit.GetComponent<Fighter>().AttackCommand(closestPlayerUnit.gameObject, true);
        }

        yield return new WaitForSeconds(Random.Range(minTimeBetweenWaves, maxTimeBetweenWaves));

        StartCoroutine(SpawnEnemyWaves());
    }
```

Apart from the manager scripts, every GameObject that constitutes an object in the level (unit, building, resource field, or resource drop) has one or more scripts that are conferring a specific functionality. Some of the most complex of these scripts are the ones attached to the units (detailed in section II. 5. Units):

➢ *III. 4. 15. Unit.cs*

The Unit class allows the unit to be moved, have stats, animations, a unit state, and current and maximum health, and also makes it possible for the unit to be selected and to be damaged and killed. It is needed by the Worker class and the Fighter classes, implementing functions used by both of them.

The movement of the units is done through the Unity Navigation System, which works by baking a NavMesh, used as the terrain on which the NavAgents can walk. All the units have a NavMeshAgent component attached, that allows them to create a path around the map of the level, avoiding any obstacles carved in the NavMesh. The units also have a NavMeshObstacle component, which is enabled only when performing a task (e.g. harvesting a resource field) and serves the purpose of preventing other units from pushing the working unit from its spot.

Every unit starts with the currentHealth set to the value of the maximumHealth and can lose part of it through the TakeDamage() function. Upon reaching 0 health, the Die() function is called, which triggers the death animation and destroys the GameObject after the death animation is finished.

The function used for setting the target destination for a unit can be found in the Annexes section, labeled as **Appendix 7.** *Function used for setting a unit's destination*.

➢ *III. 4. 16. Worker.cs*

The Worker class gives a unit the functionality for harvesting, carrying, and storing resources, as well as the ability to construct new buildings. It implements the necessary functions for each of the abilities mentioned above. It also triggers the right animation for the unit when performing each action.

When a worker unit is tasked to harvest a resource field, a cycle is started where the unit harvests the resource field up to its carry capacity, takes the harvested resource to the closest resource camp automatically (without any direct order from the player), stores the resource, and then goes back to the field, repeating the mentioned steps over and over until it is ordered otherwise or the resource field is depleted, at which point it will search for another resource field of the same type in the surroundings of the former one, resuming the cycle if another field is found, or stopping its action if not. The cycle works through the use of coroutines, waiting for each coroutine, representing a part of the cycle, to be completed, before proceeding to the next one, followed by the restart of the main coroutine upon reaching the cycle's end.

When carrying resources, the moving speed of the unit is slower, regaining its initial speed when the resource is no longer carried. The workers can also drop the carried resource, leaving it on the ground as a dropped resource GameObject, which can be picked up later by the same worker or by a different one.

When given a task, the worker unit will equip the adequate tool in its right hand, changing the tools in hand as needed, through the functions implemented in the worker class. Multiple workers can harvest a resource at the same time, or construct a building together, lowering the amount of time required for the construction to be completed.

The functions implementing this behaviour can be found in the Annexes section, labeled as **Appendix 8.** *Functions used for implementing worker's task options*.

➢ *III. 4. 17. Fighter.cs*

The Fighter class is used to give units the ability to attack and damage enemy units. Player's units having the Fighter class can be ordered to attack enemy units, fighting using a weapon until it is killed, the enemy unit is killed, or is ordered to retreat by the player.

Units with the Fighter class also gain the ability to constantly check their surroundings (over a certain distance) every second, scanning for enemy units. If any enemy units are detected, they automatically start chasing and attacking them. Moreover, if there are other allied units around when the enemy unit was detected, the initial unit which detected the enemy can alert the allied units, prompting them to attack the enemy unit too, even if it was not in their area of vision at first. If the enemy unit is running away from the attacking units, avoiding the fight, it is then chased for a certain distance, after which the chasing unit(s) return to their initial position (the one they were in before starting the chase). However, this is no longer valid when the attack was directly ordered by the player.

The constant scanning for enemy units is done through the use of the function Physics.OverlapSphere(position, radius, layerMask), which detects any colliders within the given radius distance from the given position. The GameObjects of the returned colliders are checked for the "Unit" tag, followed by the check for the unit's team. If multiple enemy units are detected at the same time, the closest one will be the one attacked.

While chasing or fighting, the fighter units will keep scanning their surroundings, checking if there are any closer enemies than the one they are chasing. If the answer is positive, the attack target is changed to the closest enemy. This behaviour helps prevent obstructions when multiple units are fighting.

Every unit can only attack again after a certain time (the attack animation's duration), after performing an attack. The damage to the enemy unit is done after the attack animation ends. This is also done through the use of a coroutine, pausing its execution for the animation's duration.

The functions implementing this behaviour can be found in the Annexes section, labeled as **Appendix 9.** *Functions used for implementing fighter's attack options.*

III. 5. Encountered Bugs

Developing a complex project can sometimes be challenging due to the appearance of bugs. Bugs are common, but having an efficient and organised style of programming helps in preventing them. While working on this demo, I tried thinking in advance of the next mechanics I had to implement, minimising the risk of bug appearance.

However, inevitably, there have been a few bugs that had to be resolved. I concentrated on playtesting as much as possible, after every new game mechanic addition, to prevent the possibility of any bugs going unnoticed. Here are some of the bugs that needed to be fixed and how I approached the situation:

- Right after finishing the worker mechanics, a bug appeared causing problems with the animation changes when a unit was ordered to harvest a resource field or pick a dropped resource when carrying another resource of a different type. To solve this, I inserted a check when ordering the harvesting or picking up of a resource that, upon detecting that the unit is already holding a resource of a different type, pauses the execution of the coroutine that harvests the field or picks up the new resource until the "let down resource" animation is completed, thus pausing the task and dropping the carried resource using the correct animation, before proceeding to carry out the ordered task, eliminating the bug.

- When implementing the game over condition, I had the game check the end condition every frame. However, since loading up a save file cleared the scene of every entity, the game ended every time I loaded the previously saved game. In order to fix this, I thought about the solution of checking the end condition every second, instead of every frame, which helped the game run more smoothly too, not having to check the number of active units every frame. But, since there was still the small possibility of loading the game right in the frame where the end condition was checked, I decided to try a different solution. By using a bool in the SaveLoadSystem, I could keep track of the save loading, and, by checking that bool inside the GameManager before checking the end game condition, no chance of the bug happening again remained.

These are only a few examples of the bugs I encountered and fixed, but I believe that delivering a bug-free application build when the product is ready is extremely important, as

bugs can heavily lower the satisfaction level for the user or player, causing an unpleasant experience, which should be avoided at all costs.

III. 6. Future Plans

As the game “Guilds” in its current state is a demo, there is a lot of space for enhancements and new additions. Having developed it with many ideas in my mind, I always tried to find an implementation that would easily allow for improvements later on. With all the core mechanics in place, adding new units and buildings will be much easier, offering huge potential for the game’s future.

Plans for some of the content I have in mind for the future versions of the game include but are not limited to:

- A tutorial level with many UI tips, to help the player understand the game’s basics.
- A transparent info box above the interaction panel displaying costs and useful information about units and buildings when the mouse cursor hovers over the button for recruitment or construction.
- More units, different than the villagers, having only the fighter function or the worker function or special units that can’t be recruited but have special abilities
- More buildings, allowing for building a more complex base and unlocking upgrades and new abilities for available units.
- Hero units that are unique in the level and have better stats than the rest of the units, having an inventory and ability system, which can turn the tide in the battles.
- New resource management system, where resources depend much more on the camp they were stored into, and having to be carried by villagers to the construction site in order to be used.
- Better AI for the enemy team, offering it the capability of constructing buildings and recruiting units on its own, as well as organizing attacks when the number of fighter units has grown large enough.

- More levels with different maps, providing more tactical planning opportunities and challenging the player to try different playstyles and approaches for achieving victory.
- Custom levels, where the player can choose to be part of a team with an allied AI and fight against a team of AI-controlled enemies on a map of his/her choosing.
- Multiplayer features: the ability to play online with one or more friends, teaming up against the AI-controlled enemy, or fighting against each other in a custom level.
- General improvements to the UI, graphics, and audio.

I believe that, with enough work, “Guilds” has the potential to become a great RTS game.

IV. Conclusion

All things considered, I believe “Guilds” is a functional game demo showcasing a potential successful addition to the constantly growing video game industry, fitting the modern genre trends well, and offering a fun and engaging experience of real-time strategic planning for any kind of player.

Being configured in a way that allows for easy expansion of the game’s core features, new levels, units, and buildings can be added with ease in future versions, providing extensive content and varied level designs with multiple levels of difficulty, destined to offer a pleasant experience no matter the skill level, allowing beginners to learn the basics, as well as providing a challenge for experienced players, accustomed to the RTS genre.

As playing strategy games has been shown to help our psychological abilities, I believe the genre deserves to be given more credit in the video game industry, and that more games and innovative mechanics should be developed for it, in order to further increase the benefits received from getting to play this type of games, along with the overall fun experience they provide us with, at a very small price and in many cases, even for free.

Using the Unity game engine has once again proved to be the right choice, assisting me to implement sophisticated game features, characteristic for real-time strategy genre with ease, as well as helping me to create easy-to-understand design patterns which can easily be used to improve the current game and add more content without causing conflicts or other problems due to game mechanics implementation flaws.

As stated in the beginning, all of the assets used in this project were found on the Unity Asset Store. The most notable asset acquisitions taken from there and implemented in the “Guilds” game demo are:

- The villagers’ models and animations (basic motions, tasks performing, and combat), along with the models used for their tools, which were made by Kevin Iglesias[5], who kindly agreed to lend me one of his paid animations packs, in order to help me in the development of this game project.
- The terrain of the map was designed using the Environment Pack: Free Forest Sample[6]
- The UI was created with the help of the Fantasy Wooden GUI Free Pack[7]
- The background music audio was provided by BGM - Asset Store[8]

Bibliography

[1] Real-Time Strategy Game Training: Emergence of a Cognitive Flexibility Trait https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0070350

[2] Personal research form regarding the benefits of playing real-time strategy video games

[3] Left-sided or right-sided minimap? https://strawpoll.com/ozo7drgpw/r

[4] Unity User Manual https://docs.unity3d.com/Manual/

[5] Kevin Iglesias Independent Animator https://www.keviniglesias.com/

[6] Forest Environment https://assetstore.unity.com/packages/3d/vegetation/environment-pack-free-forest-sample-168396

[7] Fantasy Wooden GUI https://assetstore.unity.com/packages/2d/gui/fantasy-wooden-gui-free-103811

[8] BGM - Asset Store https://assetstore.unity.com/publishers/9381

All links were last accessed on 09/06/2021.

Annexes

Appendix 1. *Functions used for implementing drag selection*

```
    private void DragSelect()
    {
        Ray selectionCornerRay;
        RaycastHit selectionCornerHit;

        Vector3[] selectionCorners = { upperLeftCorner, upperRightCorner, lowerLeftCorner,
lowerRightCorner }; //In this order!
        Vector3[] selectionMeshVertices = new Vector3[5];
        //First vertex is at camera's position
        selectionMeshVertices[0] = Camera.main.transform.position;

        for (int i = 0; i < 4; i++)
        {
            selectionCornerRay = Camera.main.ScreenPointToRay(selectionCorners[i]);

            if (Physics.Raycast(selectionCornerRay, out selectionCornerHit, 1000f,
LayerMask.GetMask("TerrainBase")))
            {
                selectionMeshVertices[i + 1] = selectionCornerHit.point;
            }
        }

        //We generate a mesh with 5 vertices: one at the camera's position and the others being our
selection corners
        Mesh selectionMesh = GenerateSelectionMesh(selectionMeshVertices);

        //We add a collider to our generated mesh so we can detect the units inside
        MeshCollider selectionBox = gameObject.AddComponent<MeshCollider>();
        selectionBox.sharedMesh = selectionMesh;
        selectionBox.convex = true;
        selectionBox.isTrigger = true;

        //We destroy the collider after a very short time so it will only detect
        //the units that are currently present inside it
        StartCoroutine(DestroySelectionBoxMesh(selectionBox));
    }

    private Mesh GenerateSelectionMesh(Vector3[] verts)
    {
        int[] tris = { 1, 3, 2, 2, 3, 4, 0, 2, 4, 3, 1, 0, 0, 1, 2, 4, 3, 0 }; //!
        //Each group of 3 numbers represent the vertice indexes for a corresponding triangle
        //Meshes are made of triangles which are only visible from one side
        //The order of the triangles' vertices has to be clockwise for a side to be drawn

        Mesh selectionMesh = new Mesh();
        selectionMesh.vertices = verts;
        selectionMesh.triangles = tris;

        return selectionMesh;
    }

    private void OnTriggerEnter(Collider other)
    {
        Unit unit = other.gameObject.GetComponent<Unit>();
        if (unit != null)
```

```
        {
            if (!selectedUnits.Contains(unit) && unit.unitStats.unitTeam == Team.PLAYER)
            {
                if (doubleClicked && unit.unitStats.unitName != lastClickedUnit.unitStats.unitName)
//Double click selection check
                    return;

                selectedUnits.Add(unit);
                unit.SetSelected(true);
            }
        }
        else
        {
            Building building = other.gameObject.GetComponent<Building>();
            if (building != null)
                selectedBuilding = building;
        }
    }

    private IEnumerator DestroySelectionBoxMesh(MeshCollider selectionBox)
    {
        //Wait for the next fixed update so the trigger collisions will be detected and then destroy
the selection box collider
        yield return new WaitForFixedUpdate();

        Destroy(selectionBox);

        doubleClicked = false;

        if (selectedBuilding != null)
        {
            if (selectedUnits.Count == 0)
                selectedBuilding.SetSelected(true);
            else
                selectedBuilding = null;
        }

        UIManager.instance.UpdateSelectedInteractionUI();
    }
```

Appendix 2. *Functions used for implementing formation movement*

```
    private void AssignPositions(Vector3[] formationPositions)
    {
        //formationPositions has the same size as the selectedUnits list
        bool[] assignedSpots = new bool[formationPositions.Length];
        bool[] assignedUnits = new bool[formationPositions.Length];

        Vector3 currentMiddlePoint = FindMiddlePoint();

        int farthestSpotIndex;
        int closestUnitIndex;
        for(int i = 0; i < formationPositions.Length; i++)
        {
            if (formationType == Formation.FREE)
                farthestSpotIndex = FarthestSpotIndexFrom(formationPositions, assignedSpots,
currentMiddlePoint);
            else
                farthestSpotIndex = i;
```

```
            closestUnitIndex = ClosestUnitIndexTo(formationPositions[farthestSpotIndex],
assignedUnits);

            assignedSpots[farthestSpotIndex] = true;
            assignedUnits[closestUnitIndex] = true;


unitSelection.selectedUnits[closestUnitIndex].MoveToLocation(formationPositions[farthestSpotIndex]);
        }
    }
```

```
    private Vector3[] SquareFormation(Vector3 targetCenterSpot, bool faceTargetPosition = true,
float yRotation = 0f)
    {
        Vector3[] squareFormationPositions = new Vector3[unitSelection.selectedUnits.Count];

        Quaternion relativeRotation;
        if (faceTargetPosition)
            relativeRotation = Quaternion.Euler(
                0f, YAnglesFromPointToTargetPoint(FindMiddlePoint(), targetCenterSpot), 0f
                ); // Here we get the rotation on the y axis between the previous middle point and
the new target spot
        else
            relativeRotation = Quaternion.Euler(0f, yRotation, 0f);

        // We turn the formation in a perfect square, then we add part of the remainder to the
square sides
        // as much as they fit, and afterwards, we put the ones left at the back of the lines.
        int currentPos = 0;
        int numberOfUnitsInSelection = unitSelection.selectedUnits.Count;
        int sideOfSquare = (int)Mathf.Sqrt(numberOfUnitsInSelection);
        int notInSquare = numberOfUnitsInSelection - sideOfSquare * sideOfSquare;
        int fullLines = sideOfSquare;
        int fullColumns = sideOfSquare + notInSquare / sideOfSquare;
        int behindFormation = notInSquare % sideOfSquare;
        int totalLines = behindFormation > 0 ? fullLines + 1 : fullLines;
        float offsetUpDown, offsetLeftRight;

        Vector3 targetCenterSpotFlat = new Vector3(targetCenterSpot.x, 0f, targetCenterSpot.z);

        if (startFromMiddleLine)
            // we start assigning positions from the center outwards
            offsetUpDown = fullLines % 2 == 1 ? 0f : -0.5f;
        else
            //we start assigning positions starting from the front row
            offsetUpDown = (float)totalLines / 2 - 0.5f;
        for(int i = 0; i < fullLines; i++)
        {
            if(startFromMiddleColumn)
                // we start assigning position from the middle outwards
                offsetLeftRight = fullColumns % 2 == 1 ? 0f : -0.5f;
            else
                //we start assigning positions from left to right
                offsetLeftRight = (-1) * ((float)fullColumns / 2 - 0.5f);
                for (int j = 0; j < fullColumns; j++)
            {
                squareFormationPositions[currentPos] = UnitPositionInFormation(
                    targetCenterSpotFlat, relativeRotation, offsetUpDown, offsetLeftRight
                    );

                if(startFromMiddleColumn)
```

```
                    offsetLeftRight = offsetLeftRight < 0f ? -offsetLeftRight : -offsetLeftRight -
1f;
                else
                    offsetLeftRight++;
                currentPos++;
            }
            if (startFromMiddleLine)
                offsetUpDown = offsetUpDown < 0f ? -offsetUpDown : -offsetUpDown - 1f;
            else
                offsetUpDown--;
        }
        // And now the ones behind who were not enough for a full line or column
        if (behindFormation > 0)
        {
            if (startFromMiddleLine)
                offsetUpDown = (-1) * ((float)totalLines / 2 - 0.5f);
            // else: it will already be at the right offset because it was lowered in the last
for-loop iteration
            if(startFromMiddleColumn)
                // we start assigning position from the middle outwards
                offsetLeftRight = behindFormation % 2 == 1 ? 0f : -0.5f;
            else
                //we start assigning positions from left to right
                offsetLeftRight = (-1) * ((float)behindFormation / 2 - 0.5f);
            for (int j = 0; j < behindFormation; j++)
            {
                squareFormationPositions[currentPos] = UnitPositionInFormation(
                    targetCenterSpotFlat, relativeRotation, offsetUpDown, offsetLeftRight
                    );

                if(startFromMiddleColumn)
                    offsetLeftRight = offsetLeftRight < 0f ? -offsetLeftRight : -offsetLeftRight -
1f;
                else
                    offsetLeftRight++;
                currentPos++;
            }
        }

        return squareFormationPositions;
    }
```

```
// This function will return the angle between the vector starting from a center point and pointing
upwards and another point (like numbers on a clock, the rotation starting at 0 degrees - at 12
o'clock)
    private float YAnglesFromPointToTargetPoint(Vector3 originPoint, Vector3 targetPoint)
    {
        float distToTarget = Vector3.Distance(originPoint, targetPoint);
        Vector3 differenceVector = targetPoint - originPoint;
        differenceVector.y = 0f;

        float angleBetween = Vector3.Angle(Vector3.forward * distToTarget, differenceVector);

        if (targetPoint.x > originPoint.x)
            return angleBetween;
        else
            return 360f - angleBetween;
    }

    private Vector3 UnitPositionInFormation(Vector3 targetCenterSpotFlat, Quaternion
relativeRotation, float offsetUpDown, float offsetLeftRight)
    {
```

```
        Vector3 targetUnitPositionNR = new Vector3(
                    targetCenterSpotFlat.x + offsetLeftRight * unitSize * distanceMultiplier,
                    0f,
                    targetCenterSpotFlat.z + offsetUpDown * unitSize * distanceMultiplier
                    );

        // Now we rotate the difference vector between the current unit position and the formation
center point based on the rotation obtained from comparing the target position to the previous
position or given from yRotation if faceRotation is set to false
        Vector3 targetUnitPositionRotated = targetCenterSpotFlat + relativeRotation *
(targetUnitPositionNR - targetCenterSpotFlat);

        return new Vector3(targetUnitPositionRotated.x,
groundTerrain.SampleHeight(targetUnitPositionRotated), targetUnitPositionRotated.z);
    }
```

Appendix 3. *Functions used for managing player-owned resources*

```
    public void AddResources(int amount, ResourceType resourceType)
    {
        switch (resourceType)
        {
            case ResourceType.FOOD:
                currentFoodAmount += amount;
                break;
            case ResourceType.WOOD:
                currentWoodAmount += amount;
                break;
            case ResourceType.GOLD:
                currentGoldAmount += amount;
                break;
        }
    }

    public bool UseResources(ResourceCost resourceCost, bool check)
    {
        if (currentFoodAmount >= resourceCost.foodCost)
        {
            if (currentWoodAmount >= resourceCost.woodCost)
            {
                if (currentGoldAmount >= resourceCost.goldCost)
                {
                    if (!check)
                    {
                        currentFoodAmount -= resourceCost.foodCost;
                        currentWoodAmount -= resourceCost.woodCost;
                        currentGoldAmount -= resourceCost.goldCost;
                    }
                    return true;
                }
                else
                    UIManager.instance.ShowScreenAlert("Not enough gold... " + resourceCost.goldCost
+ " gold needed!");
            }
            else
                UIManager.instance.ShowScreenAlert("Not enough wood..." + resourceCost.woodCost + "
wood needed!");
        }
        else
```

```
            UIManager.instance.ShowScreenAlert("Not enough food..." + resourceCost.foodCost + " food
needed!");

        return false;
    }
```

Appendix 4. *Functions used for implementing building construction*

```
    private void StartPreviewConstruction(GameObject constructionPrefab, GameObject buildingPrefab,
GameObject previewPrefab)
    {
        if (previewBuildingGO != null)
            Destroy(previewBuildingGO);

        Building building = constructionPrefab.GetComponent<Building>();
        if(building == null)
        {
            Debug.LogError("Construction building prefab " + constructionPrefab + " doesn't have
Building script attached");
            return;
        }

        if (!ResourceManager.instance.UseResources(building.buildingStats.buildingCost, true))
            return;

        previewBuildingGO = Instantiate(previewPrefab);
        previewBuildingGO.transform.eulerAngles = new Vector3(0f,
FaceCameraInitialPreviewRotation(), 0f);
        previewPlacementValidity = previewBuildingGO.GetComponent<PlacementValidity>();

        underConstructionBuildingPrefab = constructionPrefab;
        constructedBuildingPrefab = buildingPrefab;
        isPreviewingBuildingConstruction = true;
    }

    private void RotatePreviewBuilding()
    {
        if(Input.GetKeyDown(KeyCode.Q))
            previewBuildingGO.transform.eulerAngles -= new Vector3(0f, snapRotationDegrees, 0f);
        if (Input.GetKeyDown(KeyCode.E))
            previewBuildingGO.transform.eulerAngles += new Vector3(0f, snapRotationDegrees, 0f);
    }

    private float FaceCameraInitialPreviewRotation()
    {
        float closestSnapValue = 0f;
        float minDifference = 360f;
        float degrees = -180f;
        float difference;

        while(degrees <= 180f)
        {
            difference = Mathf.Abs(Camera.main.transform.eulerAngles.y - degrees);
            if (difference < minDifference)
            {
                minDifference = difference;
                closestSnapValue = degrees;
            }
            degrees += snapRotationDegrees;
```

```
        }
        return closestSnapValue + 180f;
    }

    private void StartConstructionForSelection()
    {
        if
(!ResourceManager.instance.UseResources(underConstructionBuildingPrefab.GetComponent<Building>().bui
ldingStats.buildingCost, false))
            return;

        GameObject inConstructionBuildingGO = Instantiate(underConstructionBuildingPrefab,
previewBuildingGO.transform.position, previewBuildingGO.transform.rotation,
PrefabManager.instance.buildingsTransformParentGO.transform);

        UnderConstruction underConstruction =
inConstructionBuildingGO.GetComponent<UnderConstruction>();

        underConstruction.constructedBuildingPrefab = constructedBuildingPrefab;
        Building building = inConstructionBuildingGO.GetComponent<Building>();
        building.SetCurrentHitpoints(0.1f);

        StopPreviewBuildingGO();

        // All the selected worker units start the construction
        foreach (Unit unit in SelectionManager.instance.selectedUnits)
            if(unit.worker != null)
                unit.worker.StartConstruction(inConstructionBuildingGO);
    }
```

Appendix 5. *Functions used for controlling the minimap*

```
    private void RotateMinimap()
    {
        //Rotate the minimap camera on the y axis to match the main camera
        if (minimapCam.transform.eulerAngles.y != Camera.main.transform.eulerAngles.y)
        {
            minimapCam.transform.rotation = Quaternion.Euler(
                minimapCam.transform.eulerAngles.x, Camera.main.transform.eulerAngles.y,
minimapCam.transform.eulerAngles.z
                );
        }
    }

    private void ShowMainCameraFrustum()
    {
        Ray topLeftCornerRay = Camera.main.ScreenPointToRay(new Vector3(0f, 0f));
        Ray topRightCornerRay = Camera.main.ScreenPointToRay(new Vector3(Screen.width, 0f));
        Ray bottomLeftCornerRay = Camera.main.ScreenPointToRay(new Vector3(0, Screen.height));
        Ray bottomRightCornerRay = Camera.main.ScreenPointToRay(new Vector3(Screen.width,
Screen.height));

        Vector3 topLeftCornerPoint = new Vector3(0f, 0f, 0f);
        Vector3 topRightCornerPoint = new Vector3(0f, 0f, 0f);
        Vector3 bottomLeftCornerPoint = new Vector3(0f, 0f, 0f);
        Vector3 bottomRightCornerPoint = new Vector3(0f, 0f, 0f);

        if (mapBaseCollider.Raycast(topLeftCornerRay, out RaycastHit topLeftCornerHit, 1000f))
            topLeftCornerPoint = topLeftCornerHit.point;
```

```
        if (mapBaseCollider.Raycast(topRightCornerRay, out RaycastHit topRightCornerHit, 1000f))
            topRightCornerPoint = topRightCornerHit.point;

        if (mapBaseCollider.Raycast(bottomLeftCornerRay, out RaycastHit bottomLeftCornerHit, 1000f))
            bottomLeftCornerPoint = bottomLeftCornerHit.point;

        if (mapBaseCollider.Raycast(bottomRightCornerRay, out RaycastHit bottomRightCornerHit, 1000f))
            bottomRightCornerPoint = bottomRightCornerHit.point;

        topLeftFrustumCorner = minimapCam.WorldToViewportPoint(topLeftCornerPoint);
        topRightFrustumCorner = minimapCam.WorldToViewportPoint(topRightCornerPoint);
        bottomLeftFrustumCorner = minimapCam.WorldToViewportPoint(bottomLeftCornerPoint);
        bottomRightFrustumCorner = minimapCam.WorldToViewportPoint(bottomRightCornerPoint);

        topLeftFrustumCorner.z = 0f;
        topRightFrustumCorner.z = 0f;
        bottomLeftFrustumCorner.z = 0f;
        bottomRightFrustumCorner.z = 0f;
    }

    public void OnPostRender()
    {
        GL.PushMatrix();
        {
            if (cameraFrustumMat != null)
                cameraFrustumMat.SetPass(0);
            else
                Debug.LogError("Camera outline material not assigned.");
            GL.LoadOrtho();
            GL.Begin(GL.LINES);
            {
                GL.Color(Color.black);
                GL.Vertex(topLeftFrustumCorner);
                GL.Vertex(topRightFrustumCorner);

                GL.Vertex(topRightFrustumCorner);
                GL.Vertex(bottomRightFrustumCorner);

                GL.Vertex(bottomRightFrustumCorner);
                GL.Vertex(bottomLeftFrustumCorner);

                GL.Vertex(bottomLeftFrustumCorner);
                GL.Vertex(topLeftFrustumCorner);
            }
            GL.End();
        }
        GL.PopMatrix();
    }
```

Appendix 6. *Functions implementing the save & load game-state features*

```
    public void SaveGame()
    {
        //  Creating the save file at the save path location
        FileStream fileStream = new FileStream(saveFilePath, FileMode.Create);

        // Initializing a binary formatter that will be used to serialize the save data
        BinaryFormatter binaryFormatter = new BinaryFormatter();

        // Saving game objects data
```

```
        GameSaveData gameSaveData = new GameSaveData();

        try
        {
            // Serializing the game data to the save file
            binaryFormatter.Serialize(fileStream, gameSaveData);
            Debug.Log("Game saved");
            saveFileExists.saveToBeLoaded = true;
        }
        catch(SerializationException e)
        {
            Debug.LogError("There was a problem serializing save data: " + e.Message);
        }
        finally
        {
            fileStream.Close();
        }
    }

    public void LoadGame()
    {
        // Checking to see if the save file exists
        if(!File.Exists(saveFilePath))
        {
            Debug.LogError("Save file not found in " + saveFilePath);
            saveFileExists.saveToBeLoaded = false;
            return;
        }

        // Initializing a binary formatter that will be used to deserialize the save data
        BinaryFormatter binaryFormatter = new BinaryFormatter();

        // Opening the save file located at the save path
        FileStream fileStream = new FileStream(saveFilePath, FileMode.Open);

        try
        {
            // Deserializing the game data from the save file
            GameSaveData gameSaveData = binaryFormatter.Deserialize(fileStream) as GameSaveData;
            Debug.Log("Game save loaded");

            // Setting up the game data from the save file
            StartCoroutine(SetUpLoadedGame(gameSaveData));
        }
        catch (SerializationException e)
        {
            Debug.LogError("There was a problem deserializing save data: " + e.Message);
        }
        finally
        {
            fileStream.Close();
        }
    }
```

Appendix 7. *Function used for setting a unit's destination*

```
    public void MoveToLocation(Vector3 targetPosition, bool attackMove = false)
    {
        StartCoroutine(MoveToLocationCo(targetPosition, attackMove));
```

```
    }

    public IEnumerator MoveToLocationCo(Vector3 targetPosition, bool attackMove = false)
    {
        if (target == targetPosition || unitState == UnitState.DEAD)
            yield break;

        target = targetPosition;

        if (fighter != null)
        {
            if (!attackMove)
                fighter.StopAttackMove();
            //if (unitState == UnitState.ATTACKING || !navAgent.enabled)
                //yield return StartCoroutine(fighter.StopAttackCo());
        }

        if (worker != null)
        {
            if (unitState == UnitState.WORKING || !navAgent.enabled)
                yield return StartCoroutine(worker.StopTaskCo());
        }

        yield return StartCoroutine(CheckIfImmobileCo());

        if (!navAgent.enabled || unitState == UnitState.DEAD)
            yield break;

        if ((targetPosition != target) && (fighter == null || fighter.GetAttackTarget() == null))
            yield break;

        navAgent.SetDestination(targetPosition);

        unitState = UnitState.MOVING;
    }
```

Appendix 8. *Functions used for implementing worker's task options*

```
    private IEnumerator CollectResourceCo(ResourceField resourceToCollect)
    {
        Vector3 targetResourcePosition = resourceToCollect.transform.position;

        if (unit.target == targetResourcePosition)
            yield break;

        ResourceRaw targetResourceType = resourceToCollect.resourceInfo.resourceRaw;

        Vector3 currentUnitTarget = unit.target;

        yield return StartCoroutine(unit.CheckIfImmobileCo());

        if (unit.target != currentUnitTarget)
            yield break; // unit target changed while waiting to be free for current task => task no
longer valid

        if (carriedResource.amount > 0 && carriedResource.resourceInfo !=
resourceToCollect.resourceInfo)
        {
            if (unit.unitState != UnitState.WORKING)
```

```
                yield return StartCoroutine(LetDownDropResourceCo(true));
            else
                yield return StartCoroutine(LetDownDropResourceCo(false));
            // If we go from harvesting a field to targeting another (different type), we don't lift
the current harvested resource, we just drop it
        }
        carriedResource.resourceInfo = resourceToCollect.resourceInfo;

        if (unit.target != currentUnitTarget)
            yield break; // unit target changed while getting ready for current task => task no
longer valid

        // check to see if resource was depleted while in animation state
        if (resourceToCollect == null)
        {
            // search for another field of same type in certain area around former field, and if not
found then break
            resourceToCollect = FindResourceFieldAround(targetResourceType, targetResourcePosition);
            if (resourceToCollect != null)
                targetResourcePosition = resourceToCollect.transform.position;
            else
                yield break;
        }

        yield return StartCoroutine(GoToResourceCo(resourceToCollect));

        if (unit.target != targetResourcePosition) // check to see if unit target changed while
walking towards res
            yield break;

        // check to see if resource field was depleted while walking towards it
        if (resourceToCollect == null)
        {
            // search for another field of same type in certain area around former field, and if not
found then break
            resourceToCollect = FindResourceFieldAround(targetResourceType, targetResourcePosition);
            if (resourceToCollect != null)
                CollectResource(resourceToCollect);
            yield break;
        }

        yield return StartCoroutine(HarvestResourceCo(resourceToCollect));

        if (unit.target != targetResourcePosition)
            yield break;

        if(carriedResource.amount == 0) // resource depleted while starting harvesting it
        {
            // search for another field of same type in certain area around former field, and if not
found then break
            resourceToCollect = FindResourceFieldAround(targetResourceType, targetResourcePosition);
            if (resourceToCollect != null)
                CollectResource(resourceToCollect);
            yield break;
        }

        yield return StartCoroutine(StoreResourceInClosestCampCo());

        ResourceCamp campStoredInto =
FindClosestResourceCampByType(ResourceManager.ResourceRawToType(resourceToCollect.resourceInfo.resou
rceRaw));
        if (campStoredInto != null && unit.target == campStoredInto.accessLocation)
        {
```

```
            if (resourceToCollect != null)
            {
                CollectResource(resourceToCollect);
            }
            else
            {
                // search for another field of same type in certain area around former field, and if
not found then break
                resourceToCollect = FindResourceFieldAround(targetResourceType,
targetResourcePosition);
                if (resourceToCollect != null)
                    CollectResource(resourceToCollect);
            }
        }
    }


    private IEnumerator StartConstructionCo(GameObject underConstructionBuilding)
    {
        if (unit.target == underConstructionBuilding.transform.position)
            yield break;

        yield return StartCoroutine(unit.CheckIfImmobileCo());

        yield return StartCoroutine(GoToConstructionSiteCo(underConstructionBuilding));

        if (underConstructionBuilding == null || unit.target !=
underConstructionBuilding.transform.position)
            yield break;

        yield return StartCoroutine(ConstructBuildingCo(underConstructionBuilding));
    }
```

Appendix 9. *Functions used for implementing fighter's attack options*

```
    private IEnumerator PerformAttackCo()
    {
        unit.SetImmobile(true);

        unit.unitState = UnitState.ATTACKING;
        transform.LookAt(new Vector3(attackTarget.transform.position.x, transform.position.y,
attackTarget.transform.position.z));

        animator.SetTrigger("attack");

        yield return new WaitForSeconds(unit.unitStats.attackAnimationDuration);

        if (unit.unitState == UnitState.DEAD)
            yield break;

        Unit attackTargetUnit = null;
        if (attackTarget != null)
        {
            attackTargetUnit = attackTarget.GetComponent<Unit>();
            if (attackTargetUnit == null)
            {
                Debug.LogError("Attack target GameObject " + attackTarget + " doesn't have Unit
script attached");
                yield break;
```

```
            }
        }

        if (attackTarget != null && attackTargetUnit.unitState != UnitState.DEAD && unit.unitState
== UnitState.ATTACKING)
            attackTargetUnit.TakeDamage(unit.unitStats.attackDamage);
        else
            StopAttackAction();

        unit.unitState = UnitState.IDLE;

        unit.SetImmobile(false);
    }


    private IEnumerator CheckSurroundingsCo()
    {
        yield return new WaitForSeconds(unit.unitStats.checkSurroundingsRate);
        StartCoroutine(CheckSurroundingsCo());

        // unit must not be moving and not have an attack target
        if (unit.unitState != UnitState.IDLE || attackTarget != null)
            yield break;

        // unit must not be carrying any resources
        if (unit.worker != null && unit.worker.carriedResource.amount > 0)
            yield break;

        List<Unit> enemyUnitsNearby = GetUnitsNearby(unit.unitStats.checkSurroundingsDistance,
enemyTeams);
        List<Unit> alliedUnitsNearby = GetUnitsNearby(unit.unitStats.alertDistance, alliedTeams);

        if (enemyUnitsNearby.Count == 0)
            yield break;

        Unit closestEnemyUnit = GetClosestUnit(enemyUnitsNearby);

        AttackCommand(closestEnemyUnit.gameObject, false);

        foreach(Unit alliedUnit in alliedUnitsNearby)
        {
            if (alliedUnit.fighter != null)
                alliedUnit.fighter.AttackCommand(closestEnemyUnit.gameObject, false);
        }
    }
```